Self Training Guide

DLP Module 07 – Logic Operations and Program Jumps

Introduction

The majority of DLP code works by testing the state of various digital and analog inputs using various commands. These commands have an implicit logical AND built in to them so when two or more are strung together the logic would read something like “If Input #1 is true AND Input #2 is true AND Input#3 is greater than 50%, then…..”.

While this is appropriate for most situations, sometimes the programmer will need to write a program that uses logical OR, NOR or XOR operators.

In addition to this, sometimes it is advantageous for a program to skip whole blocks of code that are not necessary, depending on the state of various inputs. The DLP language contains a few Jump commands that allow the programmer to skip sections of code depending on the state of the logic accumulator.

Before reading this document, you should have read the previous module(s) and be comfortable with the concepts discussed within. This document also assumes that you be familiar with the Q90 configuration software, and have successfully installed the DLP IDE software.

Additional details on the syntax of all DLP commands can be found in the online help.

In this document any DLP commands are presented in BLUE TYPEFACE while all DLP system variables and IO registers are in RED TYPEFACE.

The .ASM file for any DLP shown in this document is available separately.

This module contains help on:

- Logical OR (OR and OREND)
- Logical NOR (NOR and NOREND)
- Logical XOR (XOR and XOREND)
- Program Jump Statements (JUMPF, JUMPT and JUMPA)

QTech Data Systems Ltd
Unit 6 Amuri Park
404 Barbadoes Street
Christchurch
New Zealand

Phone: +64 3 3663713
Fax: +64 3 3652815
E-mail: sales@qtech.co.nz
Web www.qtech.co.nz

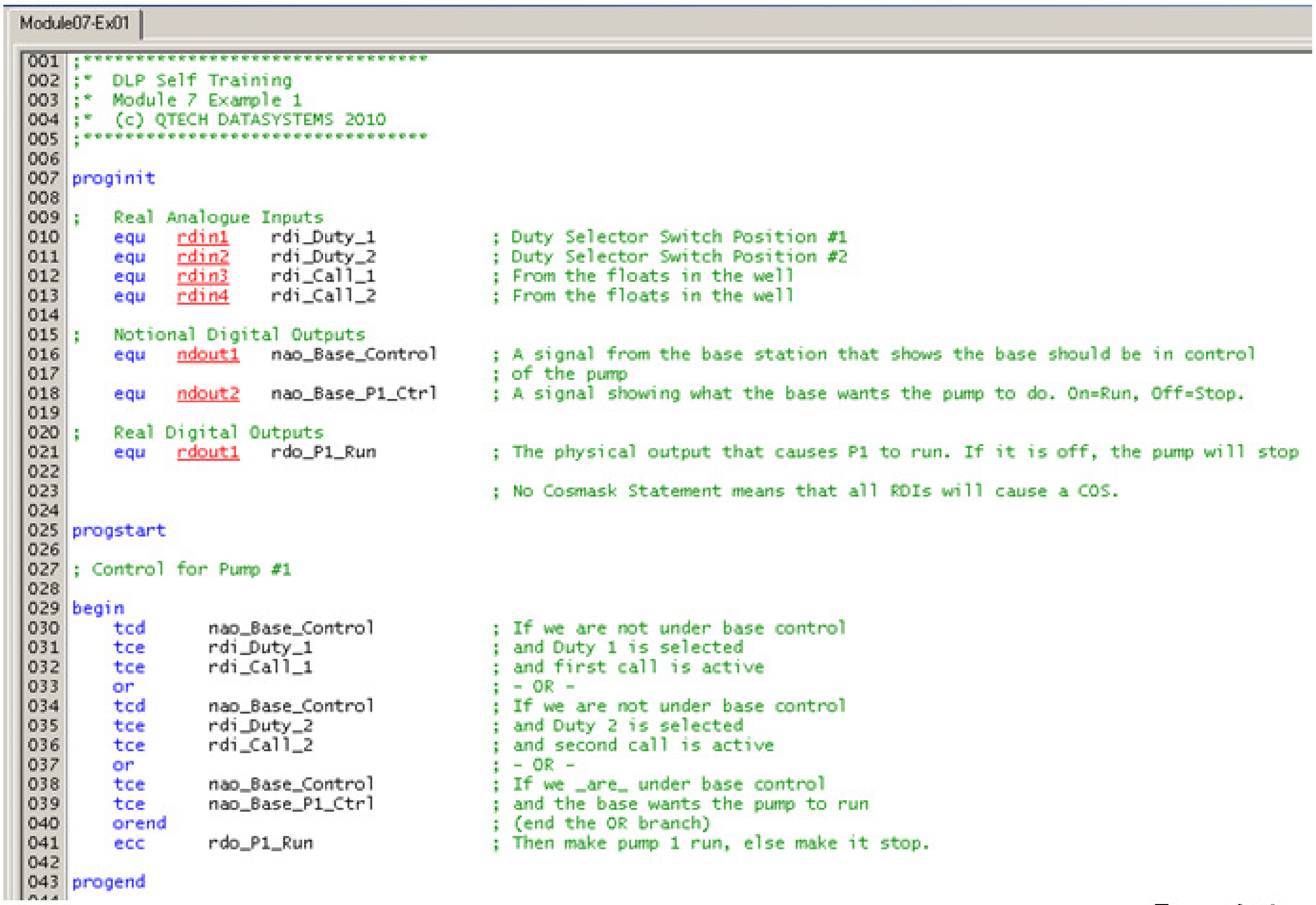
Logical OR using OR and OREND

Using a logical OR allows the DLP to take action if any of A or B or C are true. This means that a given section of code can be triggered by multiple different events.

Consider a pump at a two-pump pump station. The DLP may want to run the pump because of three different sets of circumstances. The DLP in example 1 shows a simple pump station controller that will cause Pump #1 to run if:

(Duty 1 is selected AND Call 1 is active) OR
(Duty 2 is selected AND Call 2 is active) OR
(Base Station Manual Control is selected AND Base Run #1 command is set).

Note the use of the OREND command to tell the DLP that the OR branch has finished.



```
;**********************************
;*  DLP Self Training
;*  Module 7 Example 1
;*  (c) QTECH DATASYSTEMS 2010
;**********************************

proginit

;   Real Analogue Inputs
    equ   rdin1    rdi_Duty_1                ; Duty Selector Switch Position #1
    equ   rdin2    rdi_Duty_2                ; Duty Selector Switch Position #2
    equ   rdin3    rdi_Call_1                ; From the floats in the well
    equ   rdin4    rdi_Call_2                ; From the floats in the well

;   Notional Digital Outputs
    equ   ndout1   nao_Base_Control          ; A signal from the base station that shows the base should be in control
                                             ; of the pump
    equ   ndout2   nao_Base_P1_Ctrl          ; A signal showing what the base wants the pump to do. On=Run, Off=Stop.

;   Real Digital Outputs
    equ   rdout1   rdo_P1_Run                ; The physical output that causes P1 to run. If it is off, the pump will stop

                                             ; No Cosmask Statement means that all RDIs will cause a COS.

progstart

; Control for Pump #1

begin
    tcd       nao_Base_Control               ; If we are not under base control
    tce       rdi_Duty_1                     ; and Duty 1 is selected
    tce       rdi_Call_1                     ; and first call is active
    or                                       ; - OR -
    tcd       nao_Base_Control               ; If we are not under base control
    tce       rdi_Duty_2                     ; and Duty 2 is selected
    tce       rdi_Call_2                     ; and second call is active
    or                                       ; - OR -
    tce       nao_Base_Control               ; If we _are_ under base control
    tce       nao_Base_P1_Ctrl               ; and the base wants the pump to run
    orend                                    ; (end the OR branch)
    ecc       rdo_P1_Run                     ; Then make pump 1 run, else make it stop.

progend
```

Example 1

It is important use the correct order that statements appear in such a program as the order can change the result dramatically. Example 1 shows a program that correctly uses a *(A **and** B) **or** (C **and** D) **or** (E **and** F)* arrangement.

Consider the case where the programmer needs a logic case structured like *A **and** (B **or** C)*. It might seem (incorrectly) as though this would be achieved by the following code:

```
BEGIN
        TCE        rdi_A
        TCE        rdi_B
        OR
        TCE        rdi_C
        OREND
```

This would actually give a result of (*A* ***and*** *B)* ***or*** *C* as the DLP executes in a sequential manner and the ***and*** function has a higher level of precedence than the OR funtction. To achieve *A* ***and*** *(B* ***or*** *C*), it is necessary to rearrange it so that the OR part of it comes first. The structure then becomes *(B* ***or*** *C*) ***and*** *A*.

```
BEGIN
        TCE             rdi_B
        OR
        TCE             rdi_C
        OREND
        TCE             rdi_A
```

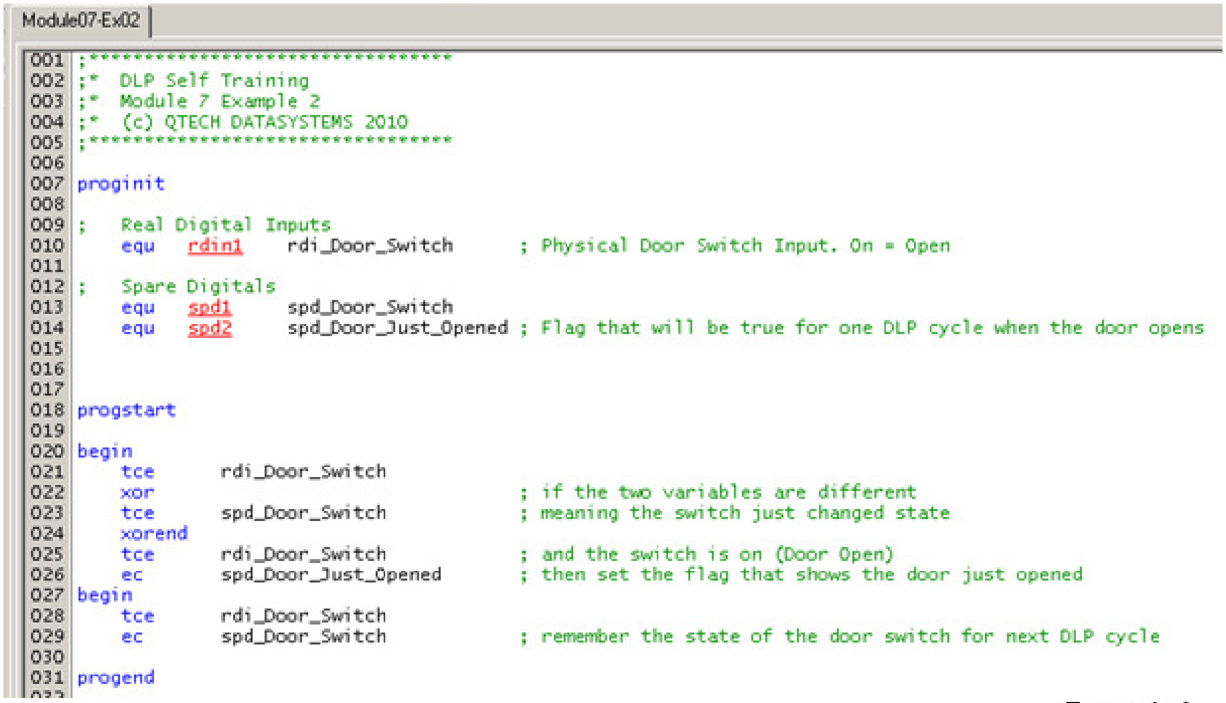
Logical NOR using NOR and NOREND

The NOR and NOREND commands work in the same manner as the OR and OREND commands, but is not commonly used.

Logical XOR (exclusive OR) using XOR and XOREND

The exclusive or (XOR and XOREND) commands work again in the same manner as OR and NOR.

XOR, however is widely used in code that is used to detect the change of state of a digital value, as shown in example 2.



```
;************************************
;*  DLP Self Training
;*  Module 7 Example 2
;*  (c) QTECH DATASYSTEMS 2010
;************************************

proginit

;   Real Digital Inputs
    equu   rdin1     rdi_Door_Switch        ; Physical Door Switch Input. On = Open

;   Spare Digitals
    equ   spd1     spd_Door_Switch
    equ   spd2     spd_Door_Just_Opened ; Flag that will be true for one DLP cycle when the door opens



progstart

begin
    tce      rdi_Door_Switch
    xor                                     ; if the two variables are different
    tce      spd_Door_Switch                ; meaning the switch just changed state
    xorend
    tce      rdi_Door_Switch                ; and the switch is on (Door Open)
    ec       spd_Door_Just_Opened           ; then set the flag that shows the door just opened
begin
    tce      rdi_Door_Switch
    ec       spd_Door_Switch                ; remember the state of the door switch for next DLP cycle

progend
```

Example 2

Program Jump Statements JUMPF, JUMPT, JUMPA

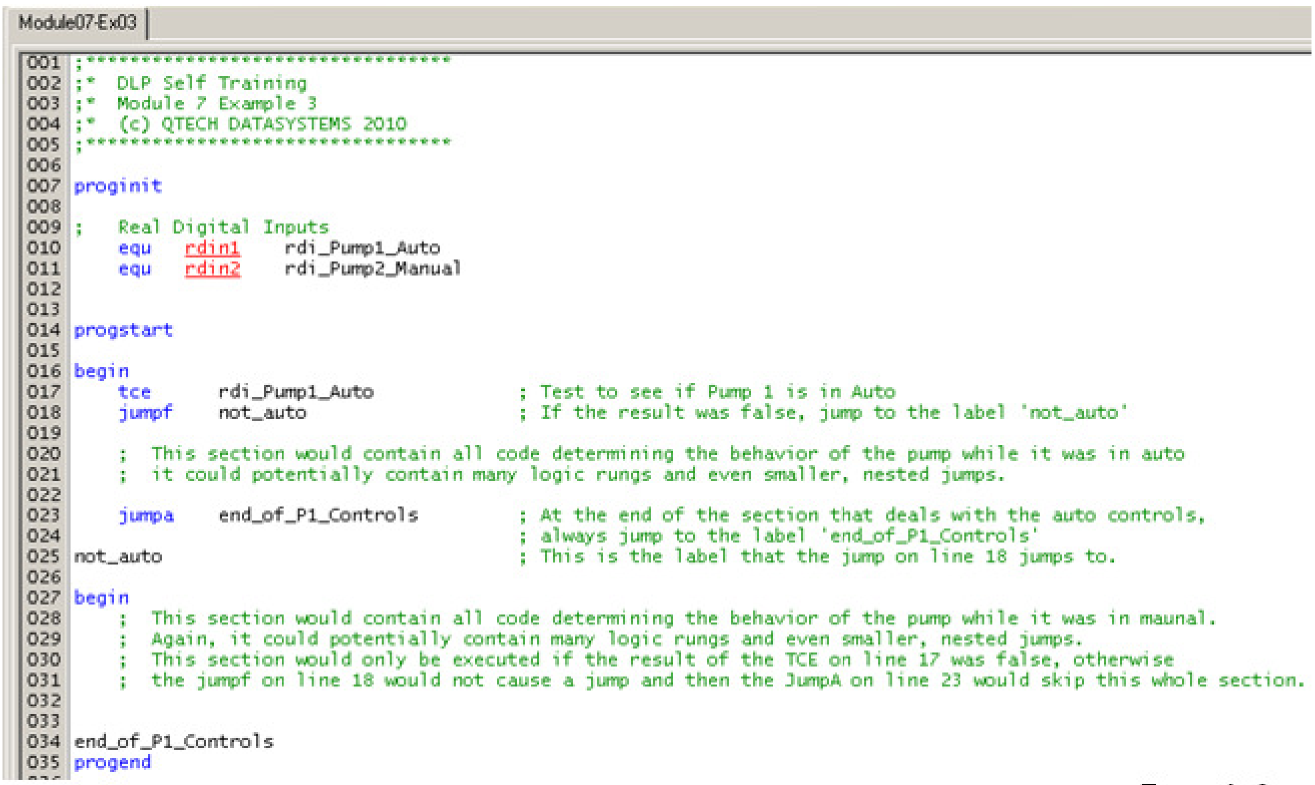
A Jump statement allows the programmer to skip certain sections of code in the DLP if specific criteria are not met. There are two reasons why we would want to do this:

- The section of code contains commands that will *always* execute such as INITANL or INITDIG
- Improving performance by jumping around sections of code that are not necessary.

The jump commands will accept two types of argument. The first is a number. This specifies the number of bytes of the compiled DLP code that the program pointer should skip ahead. This can be difficult to calculate so is not often used. The main reason the compiler supports this type of jump is that DLPs that have been extracted from an RTU will contain this type of jump, and we want the compiler to be able to compile DLPs in that format.

The second way jumps can be specified is by jumping to a “Label”. A label is a user-defined piece of text that shows the compiler where the destination of a jump command is. This way the compiler can calculate the number of bytes to jump for you. With older versions of the DLP compiler, it was necessary to define a label in the PROGINIT section of the DLP using a EQULAB command. This is no longer necessary as the compiler is now smart enough to identify these labels without predefining them. EQULAB is still supported for backwards compatibility, however.

JUMPF, JUMPT and JUMPA behave differently depending on the state of the logic accumulator. JUMPF means “Jump if the accumulator is false”, JUMPT means “Jump if the accumulator is true” while JUMPA means “Jump always” and does not care if the accumulator is true or false.


Module07-Ex03

```
001 ;**********************************
002 ;*  DLP Self Training
003 ;*  Module 7 Example 3
004 ;*  (c) QTECH DATASYSTEMS 2010
005 ;**********************************
006
007 proginit
008
009 ;   Real Digital Inputs
010     equ    rdin1     rdi_Pump1_Auto
011     equ    rdin2     rdi_Pump2_Manual
012
013
014 progstart
015
016 begin
017     tce      rdi_Pump1_Auto             ; Test to see if Pump 1 is in Auto
018     jumpf    not_auto                   ; If the result was false, jump to the label 'not_auto'
019
020     ;  This section would contain all code determining the behavior of the pump while it was in auto
021     ;  it could potentially contain many logic rungs and even smaller, nested jumps.
022
023     jumpa    end_of_P1_Controls         ; At the end of the section that deals with the auto controls,
024                                         ; always jump to the label 'end_of_P1_Controls'
025 not_auto                                ; This is the label that the jump on line 18 jumps to.
026
027 begin
028     ;  This section would contain all code determining the behavior of the pump while it was in maunal.
029     ;  Again, it could potentially contain many logic rungs and even smaller, nested jumps.
030     ;  This section would only be executed if the result of the TCE on line 17 was false, otherwise
031     ;  the jumpf on line 18 would not cause a jump and then the JumpA on line 23 would skip this whole section.
032
033
034 end_of_P1_Controls
035 progend
```

Example 3

Example 3 above shows the use of JUMPF and JUMPA in a fairly typical arrangement. The behaviour of the code shown would be similar to an If..Else kind of structure in a higher level language.

Note the use of the labels appearing in the very left hand column of the DLP. This is not necessary for the DLP to compile or function correctly, but it is an established convention that helps other programmers quickly locate and identify jump labels when inspecting the code.